The Design of a Multi-application Micro-operating System Platform in the Context of Big Data

Wenpan Mo(✉) and Zhicheng Shen

Eastcompeace Technology Co., Ltd., Zhuhai, Guangdong, China
{mowenpan,shenzhicheng}@eastcompeace.com

Abstract. In the context of big data, the security problem of smart card operating system is realized independently and controllably, For smart cards and the internet of things, a multi-application microoperating system platform is proposed. The platform's operating system is based on the virtual machine technology to support the dynamic loading, installation, deletion, and interpretation of the execution of applications, the virtual machine adopts a register-based instruction set architecture, the interpreter is faster, and the instruction optimization is easier. At the same time, the platform provides an application compiler to realize the generation of execution files running in micro-operating systems from source code parsing, compilation, and linking, which supports a variety of programming languages to develop applications, and generates register-based instruction set bytecode EF files after compilation and linking. The platform also provides a set of integrated tools integrated into application source code development, compilation, and loading of applications to virtual OS (or real OS) for application operation, testing, and debugging. Experimental results show that the proposed multi-application microoperating system has good running speed and has the ability to protect high-value and high-sensitive information, and the ability to provide high-level security protection for the device, and provide related tools such as source code development, compilation, run, testing, and debugging of applications, this improves the efficiency of application development, testing, and debugging.

Keywords: operating system · smart card · internet of things · data security

1 Introduction

With the continuous progress of communication technology and the further improvement of basic telecommunications facilities, the trend of electronization and information technology in various industries is becoming more and more obvious, and new application scenarios are being explored by the market. The application of smart cards is not only limited to traditional application fields such as finance, telecommunications, transportation, etc., but also gradually covers many fields from daily life to industrial production. In the connected world, the realization of smart connection relies on the interconnection

J. Cai et al. (Eds.): SPNCE 2023, LNICST 525, pp. 19–34, 2025.
https://doi.org/10.1007/978-3-031-73699-5_2

of devices in the front-end and cloud services and big data processing in the back-end. Once the Internet of things connects everything, it is necessary to ensure the safe access of devices, safe transmission of information, and safe storage of data. Therefore, smart cards have become an indispensable security guarantee in every aspect of our lives. The operating system is the "soul" of the information system and an important pillar of the national digital economic infrastructure. Building a "multi-application micro-operating system platform" to solve the problem of intrinsic security has become the key to the digital transformation of the economy and the development of the industrial chain.

The operating system is the basis of all software, any other software must run under the support of the operating system, without the support of the operating system, other software can not be installed and run, it plays a pivotal role in the entire computer system. In the face of various industries, the diverse needs of different users need to be solved based on different applications. Therefore, the construction of multi-application micro-operating system platform can better meet the needs of users. The implementation of the system platform, not only need to study the development of the application required source code development, compilation, running, testing, debugging and other related tools, but also need to ensure that the system platform has stability and security to ensure the normal operation of the application, and will not cause data loss or damage because of system crash or other problems. The user's data will not be attacked or stolen by malicious attackers.

2 Multi-application Micro-operating System Platforms

This section mainly introduces the overall architecture of the operating system, the analysis of key technologies, and its advantages and characteristics.

2.1 Overall Design Architecture

This paper studies a multi-application microoperating system for microprocessing, and the overall solution of related application development tools.

In Fig. 1, the platform consists of three subsystems: a compiler, a multi-application microoperating system, and an integrated development environment. The compiler supports multiple programming languages to parse and generate an abstract syntax tree, and then traverses the abstract syntax tree to generate an intermediate language, and optimizes the intermediate language, and generates bytecode files that can run on the micro operating system after linking.

Microoperating system is to interpret and execute applications, and provide a safe and reliable runtime environment for applications.

Integrated development environment (IDE) is an integrated tool for developing, compiling and loading applications to virtual OS (or real OS) for running, testing and debugging.

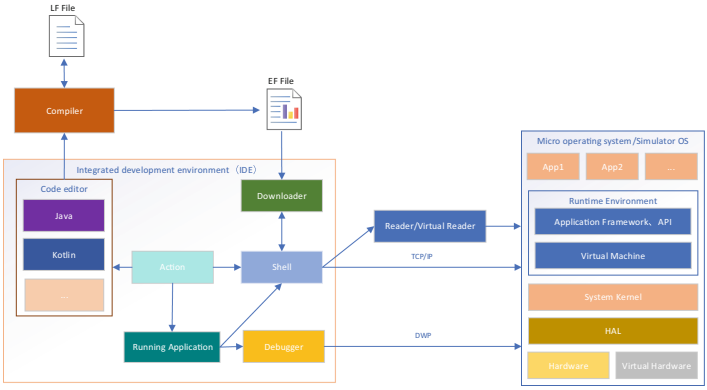

Fig. 1. Overall architecture diagram

2.2 Analysis and Elaboration of Key Technologies

The key technologies of multi-application micro-operating system platform mainly include the following five aspects: compiler, micro-operating system (MOS), object-oriented instruction set based on register, just-in-time compiler based on physical register mapping, integrated development environment.

2.2.1 Microoperating System

The main feature of the multi-application micro-operating system [10, 11, 13] is that it has a virtual machine, which separates the implementation of the application from the hardware. The application runs on a unified virtual machine, and can dynamically load, install, delete, and run the application.

As shown in Fig. 2 below, the microoperating System is divided into Application Layer, Runtime Environment Layer, System Kernel Layer, hardware abstraction layer and hardware layer from top to bottom. Among them, the runtime environment is the core of the microoperating system. The main function of the runtime environment is to interpret and execute bytecode to ensure that the runtime data of multiple applications is reliable and safe. The runtime environment is composed of virtual machine, trusted framework, resource manager, application framework, application programming interface and other components.

The design principle of microoperating system is based on the separation of software and hardware. For the same kind of hardware, it must be abstracted as a hardware interface. HAL layer is the abstract interface between hardware abstraction and VM access by application debug framework. If the microoperating system is ported to different hardware platforms, only the HAL layer code needs to be modified.

Figure 3 shows the flow chart of the virtual machine accessing the microoperating system resources. The virtual machine cannot access the microoperating system directly, but must pass through the authorization permission of the trusted framework and the agent of the resource manager to access the microoperating system resources.

The resource manager manages the microoperating system resources uniformly. The virtual machine cannot directly access the microoperating system resources through the physical address, and can only operate through the index number provided by the resource manager. Therefore, the trusted framework [3, 12] is to ensure that the data is isolated, safe and reliable when the application is running.

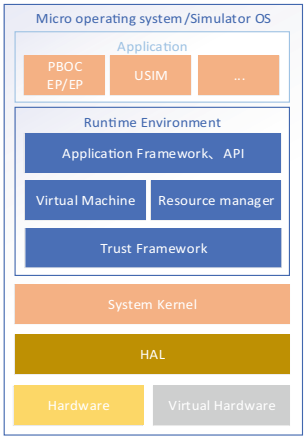

Fig. 2. Structure diagram of multi-application micro-operating system

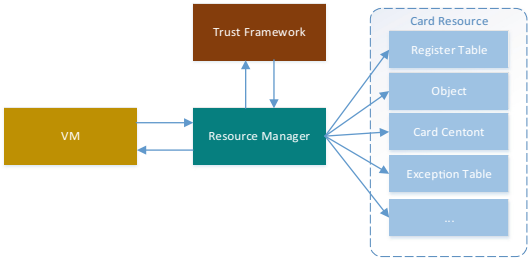

Fig. 3. Relationship diagram of core components of runtime environment

2.2.2 Compiler

The main feature of the compiler [7–9] is to support multiple programming language parsing. After the optimization and linkage of the intermediate language, the bytecode EF file based on the register instruction set is generated. As smart card is a kind of resource-constrained device, the computing power and storage space of the processor are very limited. Therefore, the symbol table is replaced by identifier linking technology in the compilation stage to solve the space occupation problem caused by symbol table and the performance loss caused by application linking when the micro-operating system loads the application.

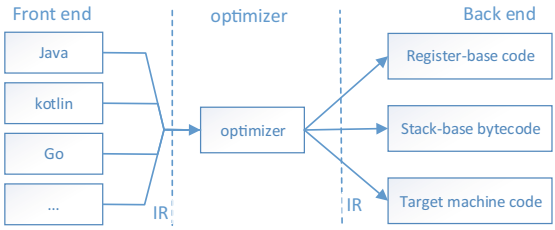

Fig. 4. Overall architecture of the compiler

The compiler is designed based on LLVM architecture (Fig. 4), which consists of three parts: front-end, compiler and back-end. The front-end and back-end use a unified intermediate code.

If you need to support a new programming language, just implement a new frontend.

If a new kind of target program needs to be generated, only a new backend needs to be implemented.

The optimization phase is a general phase, it is aimed at a unified IR (intermediate language), adding support for new programming languages does not require changes to the optimization phase.

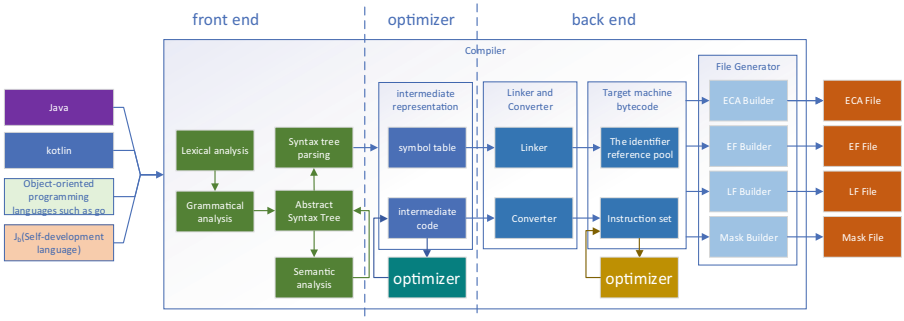

Fig. 5. Compiler technical scheme structure diagram

Figure 5, the main function of the compiler front-end is to analyze the source code lexical analysis, lexical analysis, semantic analysis, and generate abstract syntax tree; Then the abstract syntax tree is parsed to generate intermediate code. If you need to add a new programming language, you only need to implement a lexical analyzer, a syntax analyzer, and a semantic analyzer. Because this solution only supports object-oriented programming languages, the structure of the abstract syntax tree and the parsing process of the syntax tree are basically the same.

The main function of the optimizer is to optimize the intermediate code and generate new intermediate code.

The main function of the compiler backend is to generate the optimized intermediate code through the link transformer based on register instruction set, and then generate the executable file through the file generator.

2.2.3 Object-Oriented Instruction Set Based on Register

The instruction set [2] in this paper is mainly used for resource-constrained devices, such as smart card devices. The storage units and memory of smart card devices are very limited. Generally, the storage capacity of small capacity cards is not more than 64KB. The instruction set of this paper is based on the register type. Due to the limited resources of the device, this scheme will be 16 bits for a memory unit, for byte, short and other basic data types with a 16 bit memory unit, for int (int value range: -2147483648-2147483647) uses two consecutive 16-bit memory cells.

Generally, object-oriented programming languages are linked by symbols after compilation. This paper is used for resource-constrained devices. A 16-bit, two-byte unsigned unique identifier represents the link to classes, methods, domains, etc. Because strings can be infinitely long to represent classes, methods, domains and other links; Therefore, a lot of storage space is wasted.

Instruction format:
Opcode operand 1 operand 2…

An instruction by a 8 bits of operation code and multiple operands (or operand, the nop instruction is not operating).

Instructions can use virtual machine number can reach 256, register each register is 16, expressed in the adjacent two registers 32-bit data.

The opcode of an instruction is 1 byte, and the opcode values range from 0 to 255. The opcode number is 16 bits in a unit, and constants such as classes, methods are referenced using an identifier reference pool. Data types such as byte, char, short, reference can be stored in 16-bit registers, and int data types are stored using two adjacent registers.

Instruction Set Optimization Methods:
In traditional stacked instruction set, the operand of the instruction is usually stored in the stack. During the execution of the instruction, the operand is first put into the stack, the instruction logic is executed to take the operand from the stack, and the result is stored back into the stack after the instruction is executed. Therefore, the stack instruction is shorter than the register instruction, and the number of stack instructions is relatively larger for the same high-level programming language statement.

The instruction sets in this paper occupy 3-byte, 4-byte, 5-byte, and variable-length instructions. Although this kind of instruction sets meet the basic functional requirements, each instruction takes up a large amount of space, which leads to the limitation of downloadable applications on resource-constrained devices. Compared with Java Card, the space consumption of this instruction set increases by 60% (and is about 40% larger than that of Java bytecode). In this context, this paper proposes to optimize the instruction set to reduce the space consumption.

Statistical analysis of a large number of applications shows that about 99% of the methods use less than 16 registers, and the register number of instructions is used more frequently in 0–5 (70% to 80% of the time).

The Optimization Method is as Follows:

- Fixed register number: The registers (R0-R6) targeted at high frequency accesses are fixed in the instruction.
- Consolidate immediate numbers: Consolidate immediate numbers that are used only once into the instruction, reducing const assignment operations.
- Half-byte representation: Register numbers or immediate numbers are represented in half bytes, compressing the space occupied by instructions.

2.2.4 Just-in-Time Compilers Based on Physical Register Mapping

Taking advantage of the large number of registers in RISC processors, the virtual registers and physical registers used in the register-based bytecode are bound by one-to-one mapping in the bytecode interpretation phase. And compile to generate local code for execution. This scheme uses register mapping technology to compile bytecode to generate local code execution, which has high execution efficiency [1].

In this scheme, functions are divided into two categories: simple functions and complex functions. A function whose maximum number of registers is less than 10 is defined as a simple function, otherwise it is defined as a complex function. Through research, it can be found that simple functions account for more than 90% of the total number of functions (e.g. In American mobile applications, 57 functions have more than 10 registers, and the total number of functions is 886, and simple functions account for 93.57%) [5]. This indicates that on RISC architecture processors, the abundant number of physical registers is sufficient for fast allocation of virtual registers in register-based bytecode.

In this scheme, bytecode instructions are divided into two categories: basic instructions and object instructions. Array operation, object operation, static field operation, method/static method access and other instructions are collectively referred to as object instructions, and the others are basic instructions (usually compiled to generate one or two local instructions). Basic instructions are divided into two categories, namely, variable instructions and fixed instructions. The fixed instruction is that the register of operation is fixed, so the local code generated by it is also fixed, and is short and efficient, which can also be called high-speed instructions.

Taking the ARM SC300 architecture as an example, this scheme is not only suitable for ARM architecture, but also for RISC-V architecture. ARM has 13 general purpose registers from R0 to R12; The R14 (LR) register is also available if the function return pointer is stored on the stack and is recommended for storing JPC.

2.2.4.1 Register Mapping

Depending on the type of function, there are different strategies for register mapping, as follows:

1. Simple function register mapping, as shown in Fig. 6.
   a) Virtual register and physical virtual to establish a one-to-one mapping relationship;

b) The total number of allocated physical registers must be greater than the total number of virtual registers, and three physical registers should be reserved for just-in-time compilation to store temporary variables (to reduce the time consumption caused by on-site saving and recovery during compilation) and JPC use.
c) When compiling some complex object instructions, some temporary registers need to be used because of their complex logical relationships. Therefore, when compiling object instructions, the values in these physical registers selected to act as temporary registers are pushed into the stack for saving (when the reserved registers are exceeded). The value stored in the fetch stack is stored in the physical register for recovery.

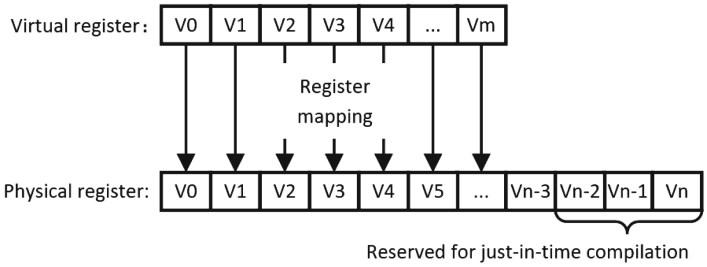

Fig. 6. Simple function register mapping

2. Complex function register mapping, as shown in Fig. 7.
   a) Firstly, physical registers and virtual registers are mapped one by one (two physical registers should be reserved for just-in-time compilation).
   b) The virtual registers that are not mapped will be stored in the stack. The area used in this part of the stack is called the overflow area.
   c) The number of allocated physical registers plus the number of overflow areas used must equal the total number of virtual registers;
   d) Similar to the simple function case, when compiling some complex object instructions, some temporary registers are needed due to their complex logical relationships. Therefore, the values in these physical registers selected to act as temporary registers are pushed onto the stack for storage (when the reserved registers are exceeded). The value stored in the fetch stack is stored in the physical register for recovery.

2.2.4.2 Just-in-Time Compiler

The virtual machine bytecode parsing process steps are as follows (Fig. 8)**:**

1. Get the instruction code and accumulate the JPC;
2. According to the instruction code, look up the bytecode function table and jump to the specified bytecode function.
3. Execute bytecode function processing logic;
4. If there is still unparsed bytecode, skip to step 1 and continue execution;

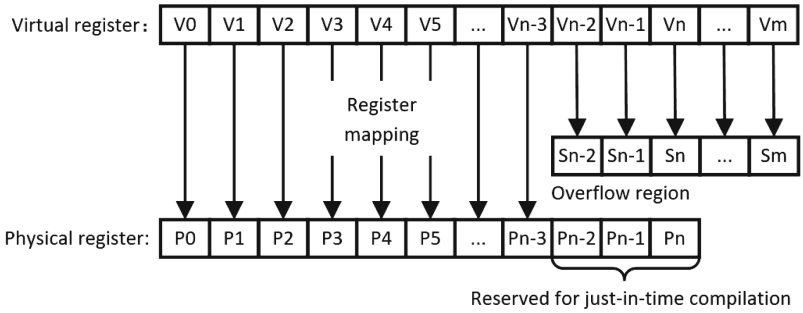

Fig. 7. Simple function register mapping

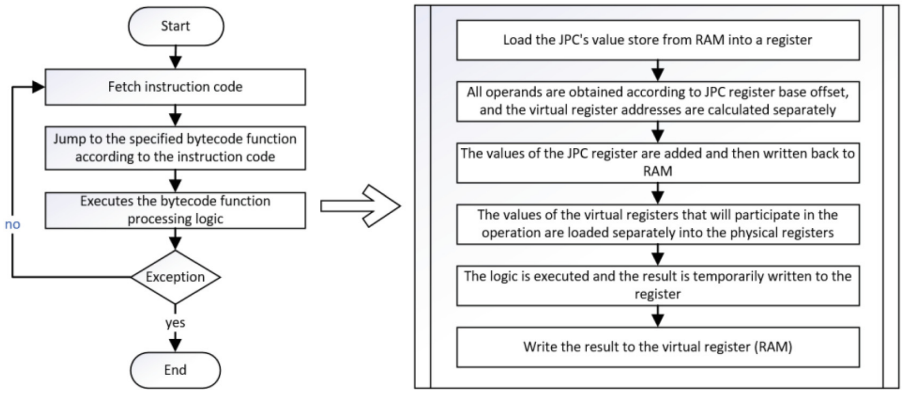

Fig. 8. Virtual machine bytecode parsing flowchart

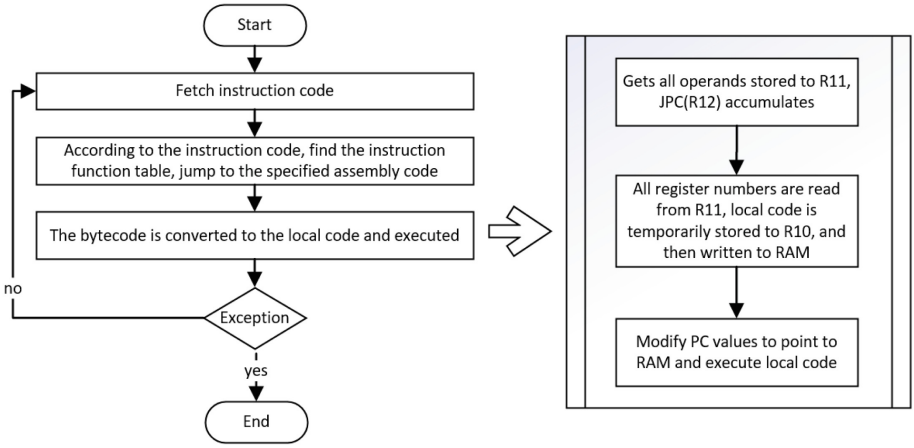

Fig. 9. Flow chart of JIT for virtual machine

5. If there is an exception, end the virtual machine execution.

The virtual machine JIT [4] process steps are as follows (Fig. 9):

1. Get the instruction code and accumulate the JPC;
2. According to the instruction code lookup function table, and jump to the specified code;
3. Get all operands stored in R10, accumulate JPC;
4. Read all register numbers from R10, assemble local code to temporarily store to R11, and then write to RAM.
5. Modify PC to point to RAM and execute the local code;
6. If there is still unresolved bytecode, skip to step 1 and continue executing;
7. If there is an exception, end the virtual machine execution.

Let's take the add instruction as an example to introduce the parse flow of JIT. The parsing process is divided into three cases: the operand is a fixed register, the currently parsed bytecode is in a simple function, and the currently parsed bytecode is in a complex function.

1. Operands are fixed registers (fixed instructions):

   Bytecode instructions: add_v0, v1, v2
   Corresponding assembly: add R0, R1, R2

2. In simple functions, add bytecode parsing flow is shown in Fig. 10:

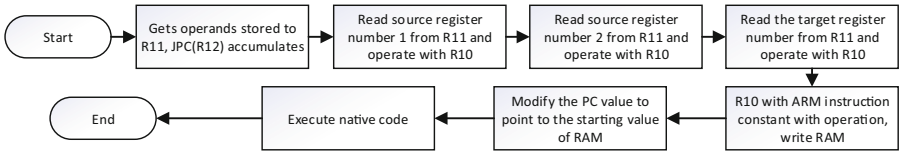

Fig. 10. Flow chart of add bytecode parsing in simple functions

2.2.5 Integrated Development Environment

Integrated development environment (IDE) is an integrated tool for developing and compiling application source code, and running the application in virtual COS for application debugging [6]. Integrated development environment is based on Eclipse plug-in development, support mainstream Eclipse version, can run on all Windows operating systems. As shown in Fig. 1, it describes the internal relationship between the main function modules and the external system.

2.3 Advantages and Characteristics of Micro-operating System

The multi-application micro-operating system platform is not only for the traditional smart card and mobile payment field, but also for the emerging Internet of things field.

It provides high-level protection for the device end, protects high-value and highly sensitive information, and provides security solutions. The micro-operating system has the following characteristics and advantages.

2.3.1 It Can Support Multiple Development Languages

The multi-application micro-operating system platform can support a variety of programming languages. The compiler can support Java, Kotlin and other static programming languages in the early stage, and can add support for c++, go, rust, Solidity and other static programming languages in the later stage. The compiler is based on LLVM architecture and has a clear hierarchical structure. It only needs to add a new programming language module support in the compiler front end.

2.3.2 Autonomous Compiler

The compiler in the multi-application micro-operating system platform is designed and developed independently, which realizes the EF file of basic register instruction set generated from source code parsing, compiling and linking. The instruction set based on register mode has faster interpretation speed and easier instruction optimization. It is designed based on LLVM architecture and consists of front-end, optimizer and back-end. Due to the modular design, the front-end and back-end use a unified intermediate code, and the same set of optimizer is used for different programming languages, which makes performance optimization simpler and more efficient.

2.3.3 Micro Operating System

When the MOS is running, the virtual machine cannot directly access the resources of the MOS. It must pass the authorization of the trusted framework and the agent of the resource manager to access the resources of the MOS.

2.3.4 The Integrated Development Environment Can Quickly Locate Defects

Integrated development environment (IDE) is an integrated tool that integrates application source code development, compilation, and loading EF files into virtual OS or physical OS to run, test, and debug applications. The environment provides powerful application code debugging functions, which can quickly locate defects, set breakpoints for single-step debugging, and view variable information in the running process.

2.3.5 Both the Compiler and the Operating System are Based on the Instruction Set of Registers

The register based instruction set of multi-application micro-operating system platform can solve the patent problem of virtual machine instruction set, and the interpretation speed is faster.

3 Experiment

This section conducts functional tests on system function development, application development, application compilation, application running, application debugging, Shell, view, and other aspects, as well as space and performance optimization tests.

3.1 Testing Environment

Development software: Keil MDK, Eclipse

Hardware: BGI CIU9872B_01, Infineon SLC36PDL352, OMNI key 3021 card reader

Operating system: Windows 10

3.2 Functional Testing

3.2.1 The Compiler

Compiler functional testing is mainly carried out from two aspects, such as compiler and micro-operating system. The compiler mainly tests whether the compiler front-end supports Java and Kotlin high-level programming languages for lexical, lexical, semantic analysis and abstract syntax tree generation, and whether the compiler back-end can generate LF, EF, ECA, MASK.C and other files, and verify whether the generated EF files can run normally in the multi-application micro-operating system.

3.2.2 Integrated Development Environment

This section examines the functions of integrated development environment from application development, application compilation, application running, application debugging, shell and other aspects.

3.2.2.1 Application Development

Check whether it supports creating application projects and applications through wizards, providing code editors for source code development, and supporting multiple high-level programming languages such as Java language and Kotlin.

3.2.2.2 Application Compilation

Check whether the compiler supports compiling and linking Java, Kotlin and other high-level programming languages to generate LF, EF, ECA files. If there is an error during the compilation process, the error information can be displayed in the “Problems” view. When double-clicking the error record, it can be located to the specified line position in the source code editor.

3.2.2.3 The Application Runs

Verify whether the EF file (application executable file) generated by the compiler can be loaded to the specified target (physical card, virtual OS, remote virtual OS, etc.) to execute the application.

3.2.2.4 Application Debugging

Verify that the application can run in debug mode and on the virtual OS. You can set breakpoints for the application code, and trace and debug the code step by step, as well as view the local variables, global variables, static variables and other information during the debugging process.

3.2.2.5 The Shell

Check that the Shell supports the following commands: /atr, /card, /close, /list-vars, /mode, /select, /send, /terminal, init-update, ext-auth, auth, upload, install, delete, card-info, set -applet, set-state, put-key, put-ketset, set-key, getcplc.

3.2.3 Microoperating System

In this section, the multi-application micro-operating system is transplanted to BADA CIU9872B_01, Infineon SLC36PDL352, and virtual OS for functional testing. The test items tested on different platforms include: virtual machine running environment, application programming interface, GP, CMCC API, SIMAPI, UICC API, one-card application interoperability, electronic wallet application interoperability, Pudong Development Development co-branded card application interoperability, American Express, etc., and all pass the test.

3.3 Performance Test

3.3.1 Performance Testing of the Application

Based on BGI CIU9872B_01 chip, this section will compare the performance of multi-application micro-operating system and Java card (based on stacked instruction set). The test precondition is that the multi-application microoperating system and the Java card operating system are transplanted to the CIU9872B_01 chip and personalized.

The financial application was selected as the test application, and the application transaction was run 50 times. The minimum transaction time, maximum transaction time and average transaction time were taken for statistics. As shown in Table 1, the overall performance is 20.4% faster than Java card.

Table 1. Application transaction performance comparison

	Min time (ms)	Max time (ms)	Average time (ms)
Java Card	739.58	779.99	741.37
MOS	611.95	654.4	615.6
Ratio	120.9%	119.2%	120.4%

3.3.2 Space and Performance Optimization Test

3.3.2.1 Space Optimization

After instruction set optimization, the space reduction is about 25%, as shown in Table 2.

The optimized space is about 17% larger than that of Java Card (usually the register instruction food occupies more than 30% more space than the stacked instruction set), as shown in Table 3.

Table 2. Comparison of the space occupied by MOS instructions before and after optimization

Comparison items	Before (byte)	After (byte)	ratio (%)
method	109493	81869	25.23%
romsize	114946	87218	24.12%

Table 3. Comparison of MOS and Java Card footprint

Comparison items	Java Card (byte)	MOS (byte)	ratio (%)
method	69767	81869	17.35%
romsize	74554	87218	16.99%

3.3.2.2 Just-in-Time Compiler Performance Testing Based on Physical Register Mapping

Based on BGI CIU9872B_01 chip, this section will compare the performance of the multi-application microoperating system before and after the JIT optimization. As shown in Table 4, the execution instruction cycles of add instructions before and after optimization are compared under different conditions. The use of fixed registers for optimized instructions improves the performance by 96%, and it improves the performance by 55% in simple functions.

Table 4. The number of instruction cycles executed by the add instruction

Conditions	before	after	ratio (%)
Fixed registers	47	2	96%
In simple functions	47	21	55
In complex functions	47	43	9%

App Performs Performance Analysis:

The financial application was selected as the test application, and the application transaction (as shown in Table 1) was run 50 times. The minimum transaction time, maximum transaction time and average transaction time were taken for statistics, as shown in Table 5. The performance of just-in-time compiler is improved by 16.6%.

Because the register number of a fixed instruction is fixed, converting to local code is also one or two local instructions to complete its logic; Therefore, it is necessary to

Table 5. Performance comparison of just-in-time compiler before and after optimization

	Min time(ms)	Max(max)	Average time
not optimized	611.95	654.4	615.6
optimization	505.85	556.4	513.3
ration	17.3%	14.9%	16.6%

add const and move as fixed instructions to improve the performance of virtual machine. According to statistics, the maximum number of registers of more than 90% functions is less than 10 (simple functions), and the estimated instruction cycle is 21, which is 55% of the original. After optimization, the overall performance of the application is improved by 16%.

4 Summary and Outlook

In the era of 5G Internet of Things, information is developing rapidly, and big data has become a hot topic. Ensuring computer network security and preventing information data leakage is an important challenge faced by current network security management. Vigorously promoting the establishment and development of information technology application innovation industry, so as to realize independent, safe and controllable alternative solutions is the need of The Times.

In this context, the smart card operating system is also facing new challenges and opportunities, and the research and implementation of the operating system is particularly necessary. The multi-application micro-operating system platform can support a variety of applications, such as finance, transportation, medical care, education, etc. These applications can share the storage space and computing resources of the smart card, and improve the utilization and efficiency of the smart card. At the same time, the multi-application smart card operating system can also provide a more secure data storage and transmission mechanism to protect the privacy and security of users. In short, the multi-application micro-operating system platform based on big data can improve the utilization and efficiency of smart cards, while ensuring the security and privacy of data. With the continuous development of smart card technology, the multi-application smart card operating system will be more widely used.

References

1. Wang Haojie, F.: Research on key technologies of program memory access analysis and optimization based on compiler technology. Tsinghua University (2021)
2. Zhang Ming, F: Research on microcontroller based on RISC-V Instruction set. Anhui University (2020)
3. Zhang Deming, F: Research on security isolation technology of embedded platform based on trusted domain. Nanjing University (2019)

4. Wenxiang Lu, F.: An improved strategy to eliminate redundant compilation based on Dalvik JIT. In: 2015 Eighth International Conference on Internet Computing for Science and Engineering (ICICSE)
5. Atsuya Sonoyama, F.: Performance study of Kotlin and java program considering bytecode instructions and JVM JIT compiler. In: 2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW)
6. Safeeullah Soomro, F.: A framework for debugging java programs in a bytecode. In: 2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)
7. Wen Yuanbo, F.: Research on cross-platform compilation technology for intelligent computing systems. University of Science and Technology of China (2022)
8. Lu Quan, F.: Analysis of the influence of different programming languages on the development of computer application software. China New Commun. (2022)
9. Wan Ping, F.: Analysis of programming language in computer application software development. Integr. Circ. Appl. (2022)
10. Chen Jianfei, F.: Research and application of embedded virtualization real-time system. Electromech. Inf. (2019)
11. Li Yunxi, F.: Research on unified architecture of multi-application mode embedded operating system. Aeronaut. Comput. Technol. (2019)
12. Zhang Linchao, F.: A brief analysis of trusted artificial intelligence system and security framework. J. Chin. Acad. Electron. Sci. (2019)
13. Xiaofeng Shang, F.: Implementation of embedded real-time operating system and application software based on smart chip. In: 2022 3rd International Conference on Electronics and Sustainable Communication Systems (ICESC)